EXPLORING THE MICROARCHITECTURAL BEHAVIOR OF AN INDUSTRIAL PROCESSOR IN THE PRESENCE OF TRANSIENT FAULTS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Catherine Mariko Trammell

May 2009

ABSTRACT

Soft errors are an ubiquitous, ever-increasing problem that will compromise future computing integrity at every echelon. Many architectures provide protection from these transient events for large arrays, such as register files and caches, but often, little is done to protect common latches from corruption, such as those used in configuration and in the pipeline. It is important that we identify the most vulnerable processor components at the latch level, so as to mitigate soft errors before they manifest in architectural state. This research evaluates the vulnerability of latches within an IBM PowerPC-based processor core. We simulate a VHDL model of the processor and use an RTX error injection methodology to inject bit-flips into the latch output nets at runtime. We then perform an analysis of the system's behavior while executing various TST applications, paying particular attention to the floating point unit, and propose solutions to increase processor robustness.

BIOGRAPHICAL SKETCH

Catherine Trammell is a native of Monterey, California. She received a B.S. in Computer Engineering from Tufts University in May 2006, and began graduate studies in the Computer Systems Laboratory in the Department of Electrical and Computer Engineering at Cornell University in August 2006. Upon receiving her M.S. from Cornell in Spring 2009 she plans to join nVidia as a hardware engineer.

For my parents, Rodney and Ann Trammell

ACKNOWLEDGEMENTS

I would like to first give thanks to Michael Gschwind and Valentina Salapura, my mentors at IBM. Their guidance has been invaluable, and without them this thesis would not have been possible. I am also in great debt to Anatoly Koyfman and Emanuel Gofman for their help with infrastructure and test case generation. Thank you to Mark Kupferschmidt, my RTX guru, and to Brian Thompto of the Z-series team for providing the error injection library. I give thanks to Thomas Roewer and Charles Wait for their help and advice with the VHDL model, and to Pia Sanda and Prabhakar Kudva for their general suggestions and support. Thank you to my manager, Ruud Haring, and the rest of his team at IBM. I give thanks to the entire Fusion group at Cornell and to my friends and family. Finally, I thank my advisor, Sally A. McKee, for the opportunities, guidance, and friendship she has given me. I owe her a great deal.

This project has been partially funded by the Lawrence Livermore National Laboratory on behalf of the U.S. Department of Energy, under Lawrence Livermore National Laboratory subcontract no. B554331.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Soft errors, also known as transient faults or single event upsets (SEUs), are a subject of increasing concern among computer architects. These errors manifest as instantaneous inversions of the logical values held in transistors, resulting in signal corruptions. For many consumers, for this to happen even once would be unacceptable, particularly for servers controlling bank transactions, voter information, national security intelligence, and countless other applications. In this chapter we discuss the causes of soft errors and the problems they can create.

1.1 Soft Error Sources

Any type of electronic noise can alter the state of a circuit, but usually this noise is generated by highly energized particles impacting the transistor substrate (Figure 1.1). The most common such high-energy particles are neutrons originating from cosmic rays and alpha particles emitted by manufacturing impurities and packaging decay. Ziegler et al. provide a comprehensive historical account of our understanding of these phenomena [20].

Cosmic rays are extraterrestrial radiation that continuously bombard the atmosphere. As they penetrate to sea level they split into various atomic particles, of which neutrons have been found to be the most problematic. Most are absorbed and lose their energy, but enough survive to create a constant influx of these particles all around us. To humans and other life forms they are harmless, but to an electronic element measured in nanometers they can be significant. Because attenuation of energy depends on the distance traveled through the atmosphere, altitude plays a substantial role in a processor's soft error rate (SER). A barrier against neutron bombardment would require several feet of concrete — clearly not a viable mainstream option.

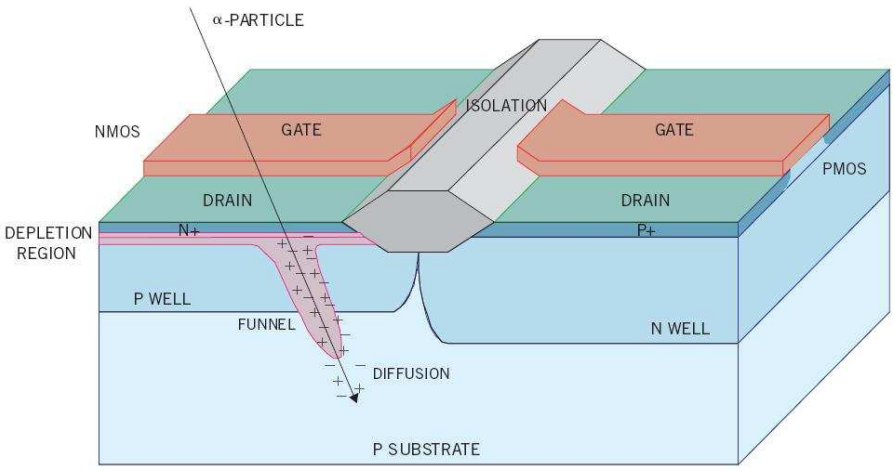

Figure 1.1: An Energized Particle Collision (courtesy of Pia Sanda)

All materials contain slight traces of radioactive substances, and so radioactive decay in electronic packaging is inevitable. Although less of a problem than cosmic rays, since measures can be taken to prevent contamination in manufacturing and since shielding is a realistic option, it still contributes to SER. Alpha particles are the most destructive result of packaging decay.

When these energized particles interact with semiconductor atoms at a particular node, the charge distribution can be disturbed sufficiently to modify the voltage level of the node's signal. The effect is temporary and does not damage the transistor (hence the term "soft" error), but the resulting mutation in a piece of data can propagate and create a lasting inconsistency. The extent of the disturbance is largely dependent on internal device characteristics such as capacitance and voltage threshold. A larger device with more capacitance is more robust against the creation of artificial current channels, and a larger voltage range requires a greater electron displacement to switch its interpreted logic level. As a result, device technology scaling is dramatically increasing observed SERs. A recent study by Karnik, Hazucha and Patel quantifies the relationship between SER and technology scaling, and they conclude that increased efforts to reduce soft errors are needed to maintain current product reliability [5]. The problem also increases

with chip area: the larger the chip, the more often (statistically speaking) an error will occur. It follows that larger systems with multiple chips are several times more vulnerable than their smaller, simpler counterparts. Although the FIT (failures in time) rate of any individual processor is relatively low, it compounds in multiprocessors, compute clusters, servers, and especially supercomputers, and becomes nontrivial for these large systems.

1.2 The Soft Error Dilemma

It is an unfortunate reality that soft error vulnerability scales with processor speed. A fast processor is likely to come with a large area, small device sizes, and the efficient utilization of resources, which means that a transient fault is not only more likely to appear but is also more likely to affect an important piece of data. In a consumer-driven market, sacrificing performance to avoid fault manifestation is often not desirable. Thus, the question involves not how to avoid soft errors altogether but rather how to pacify them once they have occurred. In order to do this it is important to understand how architectures respond in the presence of soft errors, so that targeted error mitigation mechanisms that minimize overhead yet maintain reliability and performance can be installed.

Once a transient fault has materialized in a circuit it does not necessarily translate into faulty program behavior. Often the affected resource will be idle and its result ignored anyway. Combinational logic and architectural dataflow also naturally mask many soft errors and prevent them from entering machine state. An altered ”don’t care” condition (for example, an input to an AND logic gate where the other input is already a ’0’) and a mux input that is not selected are common instances of architectural fault-masking. The percent of errors that are absorbed like this will vary depending on the architecture and the application.

In the event that an error does propagate to machine state, i.e., the register file, cache, or a state holding latch, it can become visible to the user in a variety of ways. Some may be detected by hardware via an exception or an unrecoverable failure that results in a system crash. Others may generate infinite loops and cause hangs. These are relatively harmless in the sense that the computation can be restarted and only time will be lost. A far worse scenario is the case of silent data corruption (SDC), in which an undetected error propagates to the program output and the user is given erroneous data.

1.3 Motivation

Because SER is so highly dependent on the type and design of an architecture, it is necessary to analyze the behavior of soft errors on the system in question before installing error detection and correction features. We therefore investigate the vulnerability of a new PowerPC-based IBM processor core and floating point unit, and observe how this particular processor reacts to transient fault injection. Unlike other SER studies that utilize generic architectural simulators, this research employs the VHDL model of the core design and authentic verification techniques used in product testing. Our results reflect the accurate response of a real processor in production.

Most public simulators are also limited by computation granularity. Fault injection into these tools can only be performed at the instruction level, into large arrays such as caches and register files. Injection into individual latches is not feasible. However, large arrays are easily protected by ECC (error correcting codes) and, in fact, usually are in modern architectures (ours is no exception). Latches, on the other hand, such as those in the pipeline or those that hold state and configuration, are equally vulnerable and yet often unprotected. Modeling errors that propagate into state by injecting the state itself is largely unhelpful because at that point the error will be unrecoverable, and our goal is error recovery. The more detailed simulations needed to model latch transactions are

far more time-consuming, and as a result, there is a dearth of research performed at this level. Yet it is critical that we understand what happens in detail: the frequency of propagation into machine state, the types of errors generated, and the categories of latches with the highest susceptibility.

It should be noted that the problem of soft errors is industry-wide and is in no way specific to IBM or any other company. The results presented in this paper reflect only the behavior of faults that have already manifested; they are completely independent of the rate of fault appearance. In other words we make no statement about this processor's raw likelihood of failure. This is merely a commentary on the architecture's vulnerability in the rare event that a fault should occur.

CHAPTER 2

METHODOLOGY

We model soft errors by injecting bit-flips into latches at runtime, i.e., by forcing a latch's output signal value from 0 to 1 or vice versa, and perform our experiments on the VHDL model of a PowerPC-based IBM processor core, executed with the IBM Mesa simulator. The core itself is small and multithreaded, and contains many high-performance features typical of modern processors. The Mesa simulator, combined with our RTX verification methodology, provides fine-grained error detection capabilities unavailable in most public architectural simulators, where errors are usually left undetected until they reach a processor pin. In this work we pinpoint exactly where in the microarchitecture an error first appears, and thus we detect errors much sooner after their injection. In addition, our methodology is capable of detecting fault conditions which degrade performance but do not introduce failures, such as branch mispredictions, while pin checking approaches lack this ability. In the following section we describe this methodology, implemented on the platform hierarchy illustrated in Figure 2.1.

2.1 The RTX Environment

In order to detect the appearance of a propagated error we use a tool called RTX (runtime executable), based on a framework called Fusion that drives the Mesa simulator. It is used in verification at IBM. RTX continuously monitors all simulation events such as state modification, cache transactions, and instruction issuing and retirement, and dynamically verifies this against a known correct trace of the application. Upon encountering an illegal operation or an invalid data value it reports the error and terminates the simulation. Using this environment it is possible to pinpoint the exact location in the microarchitecture that our injected fault becomes an unrecoverable error, unlike other infrastructures that can only provide high-level feedback from the user's perspective.

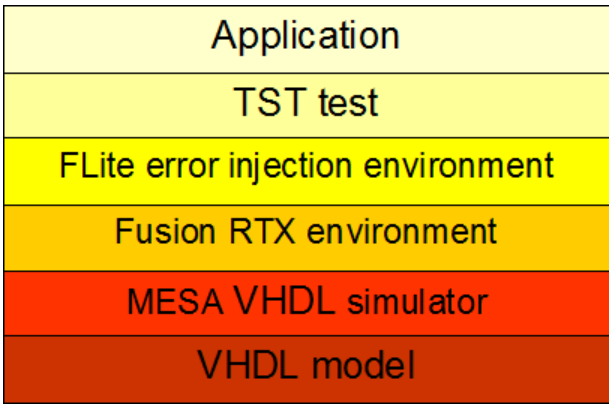

Figure 2.1: Hierarchy of Tools and Platforms

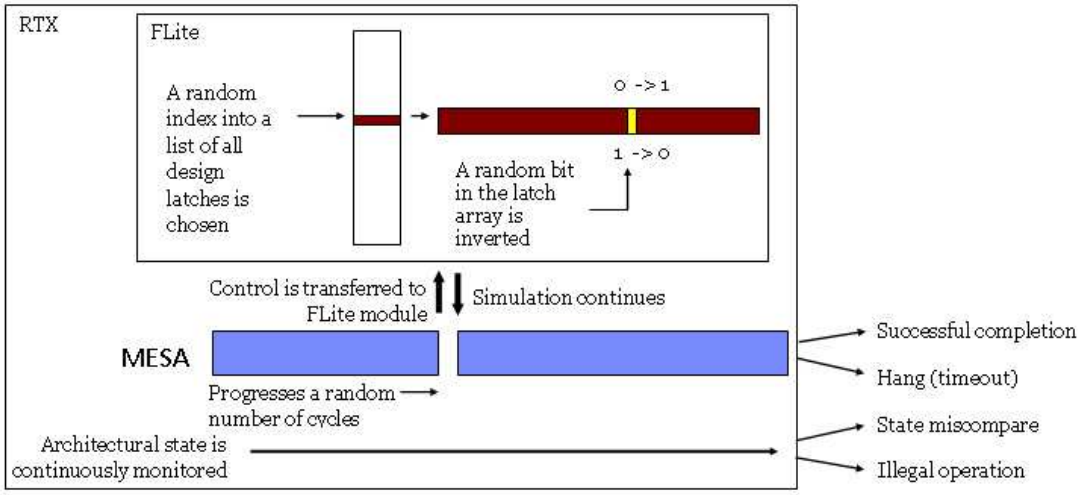

Figure 2.2: Experimental Process

To access latch signals we utilize an error injection library inherited from the IBM Z-series reliability team and adapt it to our VHDL model. The library uses a Fusion-based environment called FLite (Fusion Lite) that improves usability for RTX; all error injection takes place within this FLite framework. Before Mesa is launched, FLite initializes and instructs the simulator to suspend execution when a fault is to be injected and to temporarily yield control to the FLite module at that time. After the fault is introduced, control is passed back to the simulator. This process is shown graphically in Figure 2.2.

Within the FLite module, user-specified parameters control which latch gets injected

and at which cycle the injection is performed. The specified latch name (given as a string) is first parsed, and a handle is assigned to the corresponding RTX facility. Facilities are RTX objects that map to each VHDL net, providing a means to track and manipulate signal values in the simulator. For us they are the gateway for the dynamic introduction of bit-flips. Now because individual latches, each holding a single bit of data, are often grouped to hold multi-bit data, the term ”latch” is generically assigned to any such grouping of variable width. So in addition to a latch name we specify a bit within the group to corrupt, and our FLite module inverts the sense of that bit in the facility. To communicate the name of the latch to inject, a list of every latch name in the entire model is given in a text file. The list comprises the subset of names in the model's netlist with a common latch suffix, excluding arrays that have existing soft error protection (such as register file internals) and certain elements that are disabled during normal operation (such as ABIST, or array built-in self test). This study is thus limited to unprotected latches that are most likely to be in use at any given time. Upon FLite invocation the latch list is copied into an array of strings, and the latch that will be targeted for that simulation is chosen with a random index into this array.

More often than not, an injected error will not cause a user-visible error, and for this reason some techniques inject multiple errors into a single simulation, for the purpose of obtaining more interesting results in less time. We opt for the more realistic approach of injecting only one bit-flip per simulation and allowing the program to run to completion. This is not only representative of how soft errors are likely to materialize in practice, but also enables us to identify a solid cause-and-effect between each injected fault and resulting simulation error. We compensate the small fraction of ”interesting” runs with numbers across the board. With over 6000 latches in the simulation space consisting of over 70,000 individual latch bits, and with a range of up to 10,000 potential injection times, the number of runs required to ensure statistical significance in our results is

large: to be accurate to within 1% with 99% confidence, over 16,000 are needed per application. With the help of IBM's vast computing resources and a little automation, this was achievable.

A Perl script controls the randomization of injection parameters and supervises the job queue. The first parameter, latch name, is simply a random number between zero and the total number of latches being tested (to be used as an array index, as previously discussed). The valid range for the second parameter, latch bit, varies across latches, and therefore requires a mapping between each latch and its width. We append to our existing list of latch names the total number of bits in each latch, and the chosen index then yields both a net name and an upper bound for the latch bit parameter. Finally, we randomize the time of injection, whose range is limited to the duration of the main loop in each program, such that no fault is ever injected during the uncharacteristic phases of the code (i.e., the beginning and the end). The range is also cut prior to the end of the final iteration, with the aim to capture error propagation characteristically as well. Within the range, injection time is further restricted to always occur at the same point in a cycle, made possible because each processor cycle is actually equivalent to four simulation cycles. One additional parameter called the Fusion masterseed is always set to the same constant; this ensures that all hardware design variables such as memory latency and cache miss rate are identical across simulations and that we are always comparing apples to apples. This also ensures determinism and reproducibility in our results. All of these parameters are passed to FLite via a separate file that must be unique for each run, and thus a script keeps as many parameter files in existence as there are spawned jobs. The script submits jobs until a user-defined limit is reached, but also monitors the queue size and suspends job submission when the queue starts to get overwhelmed.

2.2 TST Test Cases

The requisite test cases for RTX have a special format known as a TST. TSTs are short instruction sequences, usually containing on the order of tens of instructions, that are used to verify hardware functionality but do not necessarily compute anything meaningful. In addition to instructions themselves, the format provides complete information as to what data is computed and what the architectural state should look like before and after each instruction. Verification is performed on VHDL models by comparing these traces to what occurs in the simulator and reporting discrepancies. Not surprisingly, TSTs are exceptionally slow to execute, which is why they ordinarily are not created with any connection to a real application, but because soft errors are data dependent it was necessary for us to create this link in our experiments.

Using the kernel source code for two major floating point applications, we created four distinct, industry-relevant TSTs. This was done primarily with the help of an IBM tool called GPro that generates TST files from assembly-based input. Assembly code extracted from the source files underwent the extensive manual process of conversion to the PowerPC architecture, followed by conversion to the GPro format, followed by a filtering of unsupported instructions and conflicting addresses, and finally GPro compilation into a TST. Because TSTs typically do not represent real applications, no infrastructure exists to make this slow transformation, and as a result, time constraints prohibited the acquisition of more than four benchmarks. Each of the applications was converted into two TSTs: one with a single thread of execution and one with four threads. The multi-threaded version in each case is simply a four-way replication of the same code on different sets of data, thus multiplying the total workload by four. Because a thorough translation of the full applications would have yielded TSTs over 800,000 lines long, we use only the kernel and execute the most significant portion of the applications in a few thousand instructions.

Table 2.1: TST Attribute Summary

	Instruction Count	CPI
QCD: 1 thread	2572	5.010
QCD: 4 threads	8876	1.488
DGEMM: 1 thread	1112	3.731
DGEMM: 4 threads	4448	1.108

Table 2.2: TST Instruction Mix

	Control	Floating Point	Integer	Memory
QCD	6%	78%	12%	4%
DGEMM	2%	55%	15%	28%

Our chosen applications for this study are QCD and DGEMM, two floating point intensive programs. A summary of their attributes is shown in Table 2.1. For each TST we list the number of instructions executed and the simulated CPI[1]as reported by RTX. QCD models lattice quantum chromodynamics and serves as our mainstream benchmark with a pretty typical workload. DGEMM, a matrix multiplier, is more of a worst-case application, with efficient pipeline utilization and above average throughput. QCD is also about twice as large as DGEMM. In Table 2.2 we provide the relative contributions of primary instruction categories found in each application, keeping in mind that loops have already been unrolled. Although targeted for a floating point study, there is no shortage of integer instructions in either TST, and they both sufficiently tax the entire processor to obtain a comprehensive set of results.

[1]Actual CPI numbers from performance testing of the DGEMM executable on a verified pre-hardware simulator were found to be 3.85779 with a single thread of execution and 1.09982 with four threads. Simulator incompatibility with the QCD executable prevented the verification of our QCD CPI numbers.

2.3 Analysis Techniques

There is no doubt that absolute error percentage is a valuable statistic. After all, users only see and care about how often they witness a failure. However, reducing the raw SER requires more knowledge about hardware status at the time of a fault. Therefore, from a design perspective we are interested not only in how often errors occur, but what the conditions surrounding each error are. One important such condition is the status of the latch at the time of corruption. If we know that the latch was not carrying sensitive data we learn nothing about how robust the hardware is. Luckily our host processor utilizes clock gating to reduce power consumption on latches that are not needed for the current computation. By probing the clock signal of the latch we inject, we can determine which latches are inactive and which at least have the potential to propagate an injected fault. We inspect a simulation waveform to see when clock periods start, to verify that probing always occurs when the signal should be at a logic '1'. Since injection always takes place at the same point in a cycle, we simply have FLite probe the clock at a constant offset from our chosen injection point. For most latches in the model, the net names for the output and clock signals differ only in the suffix, allowing us to obtain the clock net name directly from the latch name. There are a few that do not have this trait, so we implement a small fix in the model to create aliases to the clock signals in this rare case. With this modification all clock nets can be produced through string manipulation in Perl.

As mentioned previously, our VHDL model often groups multiple latches together and provides all of them with a single net name, making purely uniform error injection difficult. In our methodology we randomize injections uniformly across latches, ignoring width. This skews our results in favor of smaller latches, whose bits contribute more to overall vulnerability than they would in reality. To compensate, we weight the number of fault injections and failures seen for a particular latch by the number of bits

in the latch, so that the distribution of injections more closely resembles a uniform one. This, of course, assumes that latches will exhibit homogeneous behavior for all bits, but since they are grouped for the very reason that they share a common function, this is a fair assumption to make. A good subject of future research would be to replicate these experiments accounting for latch width, but for our purposes this seems unnecessary, particularly since small latch behavior is still of high interest and decreasing its frequency in our results is not desirable. Consequently, some results we report have been weighted to help relieve bias toward smaller latches.

In many of our experiments we account for a set of latches in the model with a hardened design style. These more robust latches are assigned to important state-holding elements in the execution, instruction, and memory management units, and in practice will be less susceptible to fault manifestation. SEU hardening of this sort can be implemented using a variety of different methods, ranging from increasing the capacitance at the transistor node (as in [6]) to adding error-correcting feedback circuitry to the latch (as in [7]). Krishnamohan and Mahapatra provide an analysis of different types of hardening techniques in [8]. Improving latch robustness invariably requires additional overhead, so these techniques are not used for every latch in the processor. We therefore consider the diversity of latch designs in our analysis of architectural reliability.

CHAPTER 3

RESULTS

In this section we outline the results of our experiments. The first set of results reflects the complete space of latches chosen for the study, while the second set extracts the aforementioned set of latches with a hardened design style. We present an overview of processor vulnerability for the worst-case scenario in which these latches will still carry faults, followed by a detailed analysis of the more likely scenario in which they will be resilient against soft errors. Finally, we conduct a different experiment designed to test SER uniformity within a single floating point latch.

3.1 Worst-Case Results

Figure 3.1 summarizes the results for each of the four TSTs, comparing the ratios of runs that pass and fail with injections into active (clocked) and idle (not clocked) latches. These results are based on pools of more than 20,000 simulations for each TST. Runs that pass are simulations that successfully complete with no anomaly detected by RTX; there is never any corrupted state found in the architecture nor any unexpected operation. Failed runs that were not clocked represent the small fraction of injections directly into state-holding latches, where RTX detects their presence immediately after injection. The plots show that even assuming every member of our hardened latch set fails, between 80% and 88% of all faults are absorbed in the architecture. The multi-threaded TSTs are more vulnerable than their single-threaded counterparts by 5% and 8% for DGEMM and QCD, respectively. QCD does not harbor fewer failures than DGEMM, despite having a larger CPI and a slightly larger percentage of injections into clock gated latches (1%). Thread-level parallelism appears to have a much greater effect on error rates than does instruction-level parallelism.

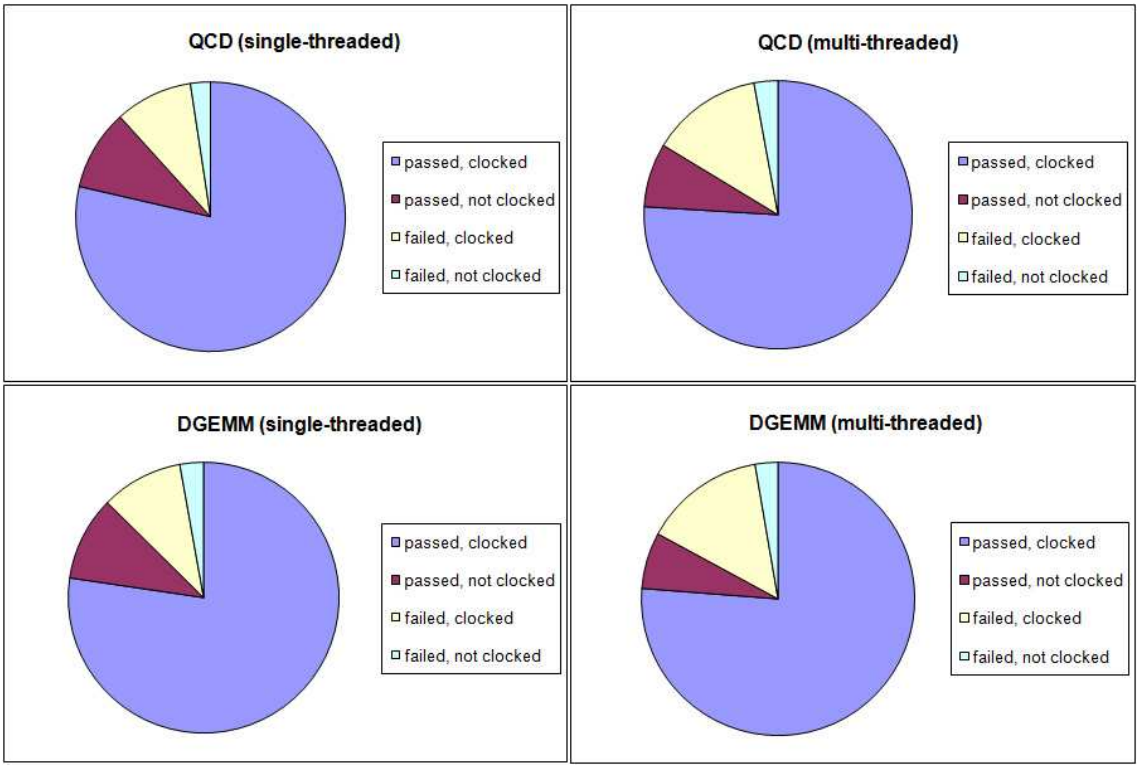

Figure 3.1: Summary of Worst-Case Results

It is important to note that even clocked runs that fail RTX may not necessarily result in program output corruption. It is possible that a corrupt state-holding element, for example, will never be used and thus never propagate its fault anywhere. However, once an error appears in state, it will remain until overwritten or until the thread completes. State errors are therefore likely to cause problems, and even within the same application will become more problematic with increased runtime or input size. Their presence greatly increases the potential for corruption and should consequently be avoided if economically feasible to do so.

We further break down the results into the major units of the processor, which are the pervasive unit (used primarily for debugging), the instruction unit, the integer execution unit, the memory management unit, the floating point unit, and the boxes unit (used for mailbox communication). Table 3.1 lists these units and their abbreviations, which we will use from this point on. Also listed in the table are the relative sizes of each

Table 3.1: Processor Units

	Function	Latches	Latch Bits
A_PCQ	Pervasive Unit	79	1690
IUB	Instruction Unit	1499	15577
A_XUQ	Integer Execution Unit	3269	29598
A_MMQ	Memory Management Unit	552	7592
F4B	Floating Point Unit	403	13468
BX	Boxes Unit	155	2129

unit, given as both the number of latches and the number of latch bits devoted to that unit. A_PCQ and BX are relatively diminutive, and so we omit them from many of our analyses. Figure 3.2 categorizes the results by the other four units embodying the processor core. The execution unit dominates the latch bit count in the processor, and so it is not surprising that it also dominates the percentage of passed and failed runs, even though our TSTs consist primarily of floating point instructions.

3.2 Representative Results

We now take the hardened latch set into account. These more resilient latches are found only in the IUB, A_MMQ, and A_XUQ, and comprise no more than 5% of each TST's pool of simulations. We filter these runs from the dataset using a Perl script, and report SERs more representative of what this architecture will exhibit in reality. The summary of this new subset of results is shown in Figure 3.3. Error rates actually improve by about 4% for all TSTs because the hardened style was assigned mostly to state-holding special purpose registers (SPRs), which almost always generate automatic failures for RTX. Figure 3.4 shows the new results organized by the four main processor components. All three units with hardened latches demonstrate lower SERs in the new results, but A_MMQ, in particular, transforms from a significant source of failures into the most

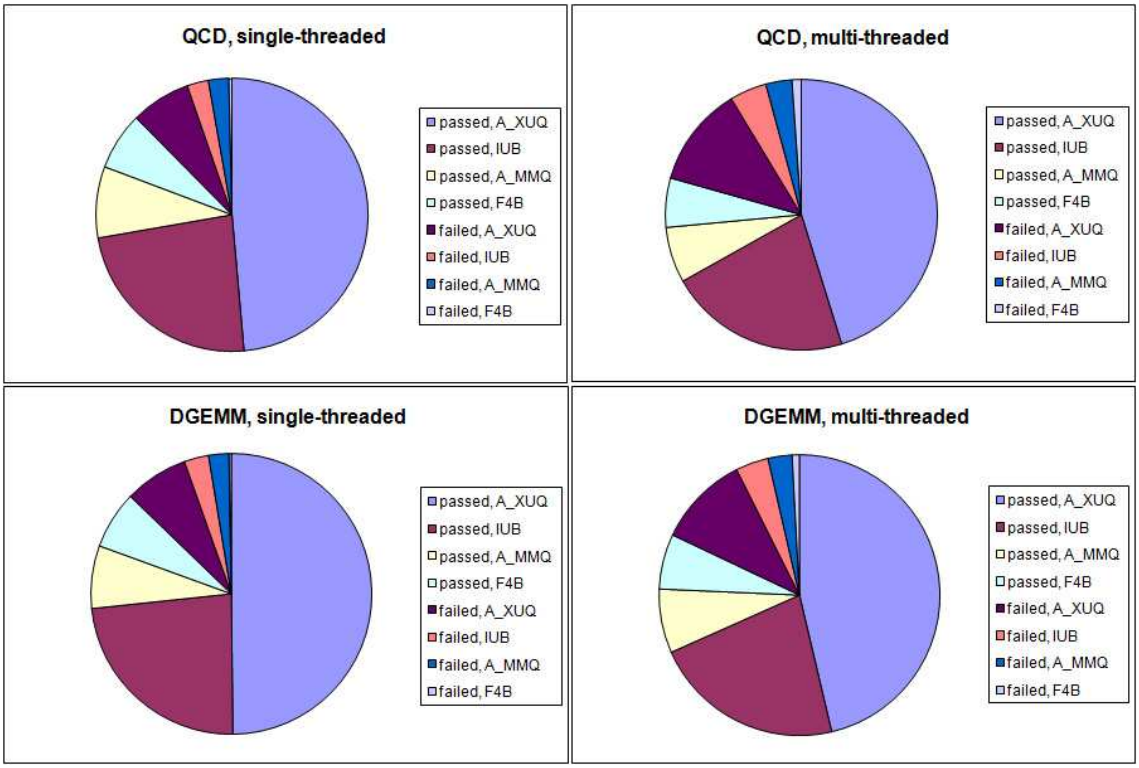

Figure 3.2: Worst-Case Results by Unit

resilient major component in the processor. Judging from the figure, this unit inherently does not have a tendency to propagate errors and is most susceptible via direct state latch particle strikes. Protecting these SPRs to resist fault manifestation in the first place can dramatically improve SERs.

3.2.1 Unit Breakdown

Since units vary widely by size, we need to isolate the results for each unit to determine its vulnerability. Figures 3.5, 3.6, 3.7, and 3.8 portray the characteristics of the IUB, A_XUQ, A_MMQ, and F4B components, respectively. IUB and A_XUQ suffer the most from soft errors, despite having many of their state-holding elements removed from their datasets. Heavy utilization of these components probably influences the SER but is not the sole contributor, since the TSTs were chosen to stress the floating point unit as well.

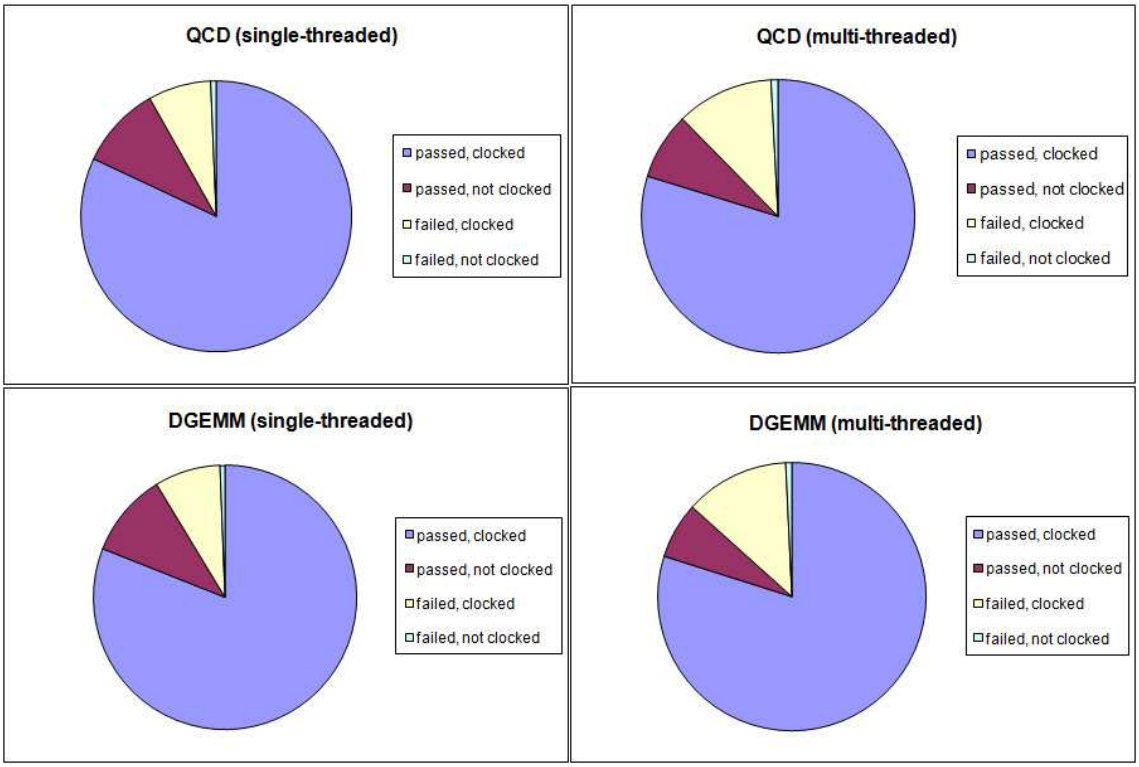

Figure 3.3: Summary of Results (No Hardened Latches)

A_MMQ not only boasts a 90% or more success rate, but it does so without a single injection into an inactive latch. The reason is that the VHDL model is still in production, and clock gating has not yet been added to the memory management unit. In fact, the floating point unit is the only component with extensive clock gating, as evidenced by the large percentage of injections into idle latches. It may not be coincidental that the floating point unit also has one of the smallest failure rates. Since our applications are primarily floating point in nature, one would expect F4B to be more vulnerable than A_XUQ, but this is not the case, and it is unlikely that the extreme disparity in clock gating schemes between the two components has no influence. Reasons for clock gating do not exclusively involve turning off elements with no effect on upstream logic; architects often wish to preserve latch data while reducing power consumption. A clock-gated latch prevents a fault from entering the feedback loop and becoming a lasting artifact, while a latch that remains unnecessarily active can still unintentionally retain

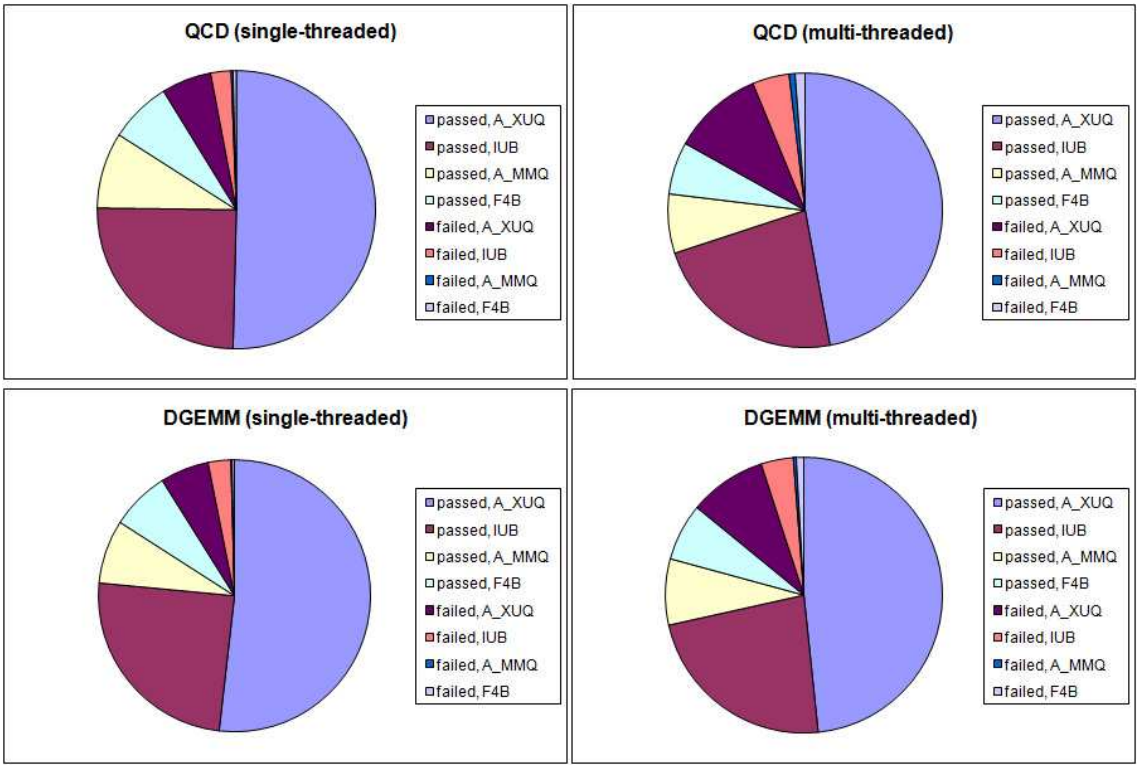

Figure 3.4: Results by Unit (No Hardened Latches)

and propagate an error. Thus it is logical that the unit with more thorough clock gating displays greater soft error resilience. Clock gating may be primarily a power-saving mechanism, but it offers data protection as well.

3.2.2 Failure Types

Fault injection produces a wide variety of errors reported by RTX. Figure 3.9 shows the most common of these errors for each TST (among injections into active latches). Idle latch injections that lead to failures are omitted because they almost universally cause IBI (instruction by instruction) miscompares of some type and are most likely state miscompares found at the point of injection. They do not provide any useful information about soft error propagation behavior. The difference between the two flavors of IBI miscompares pertains to the type of latch in which an error is found, but for our purposes

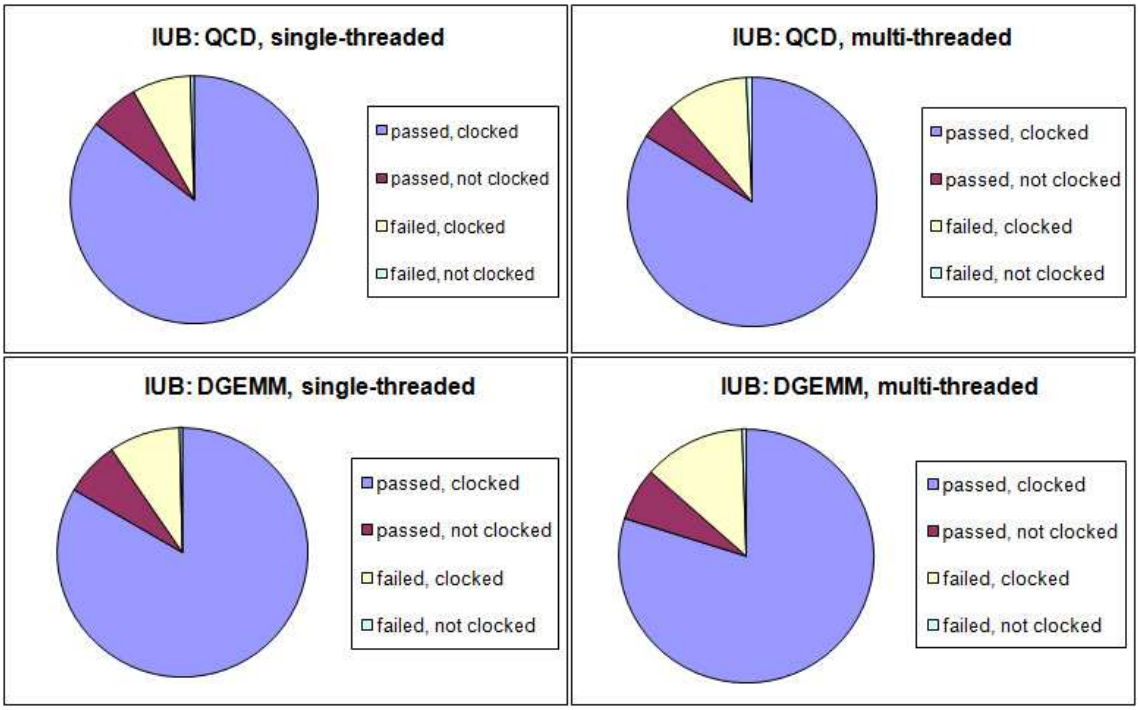

Figure 3.5: Instruction Unit Results

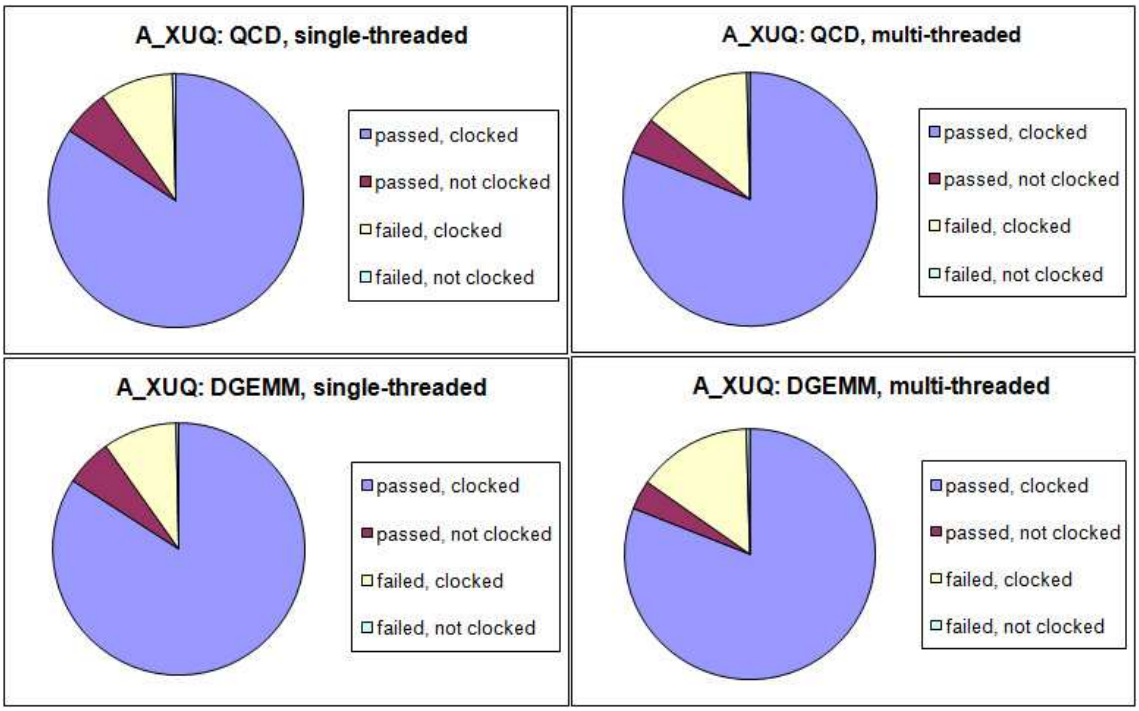

Figure 3.6: Execution Unit Results

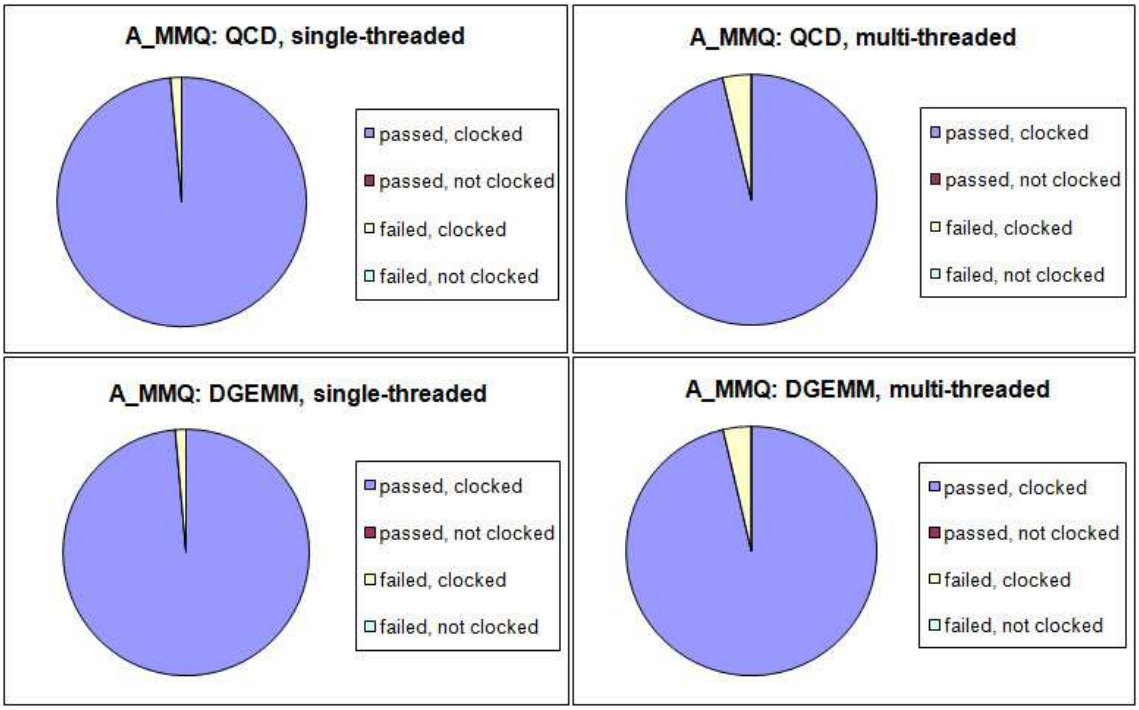

Figure 3.7: Memory Management Unit Results

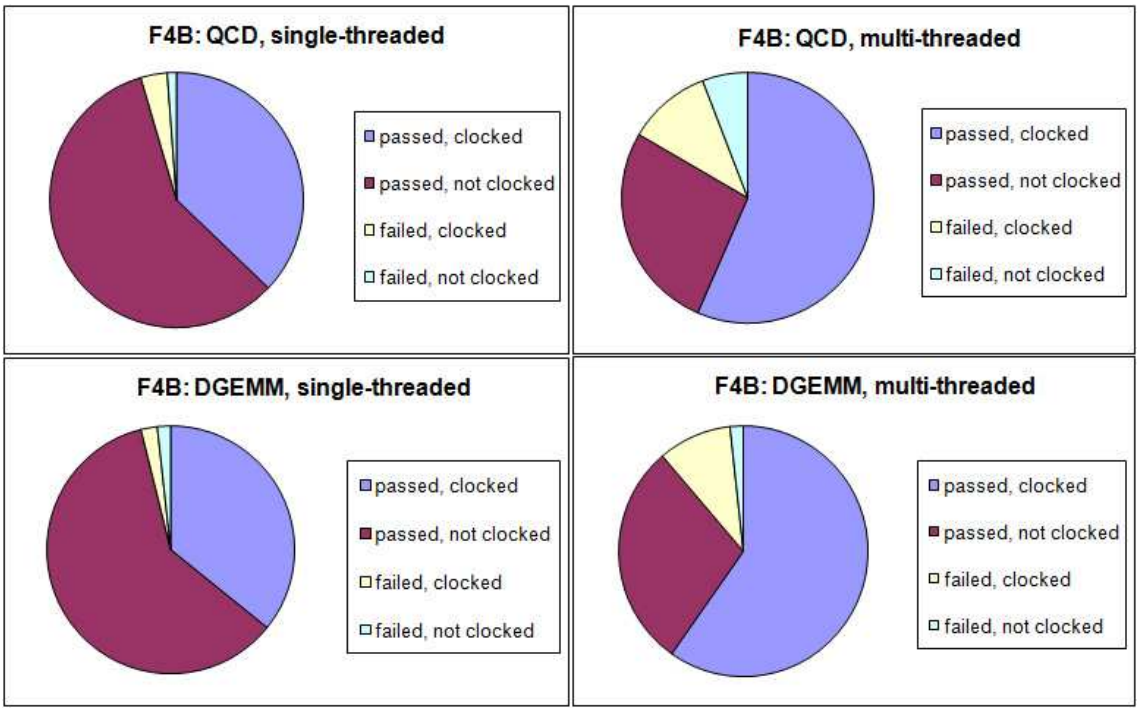

Figure 3.8: Floating Point Unit Results

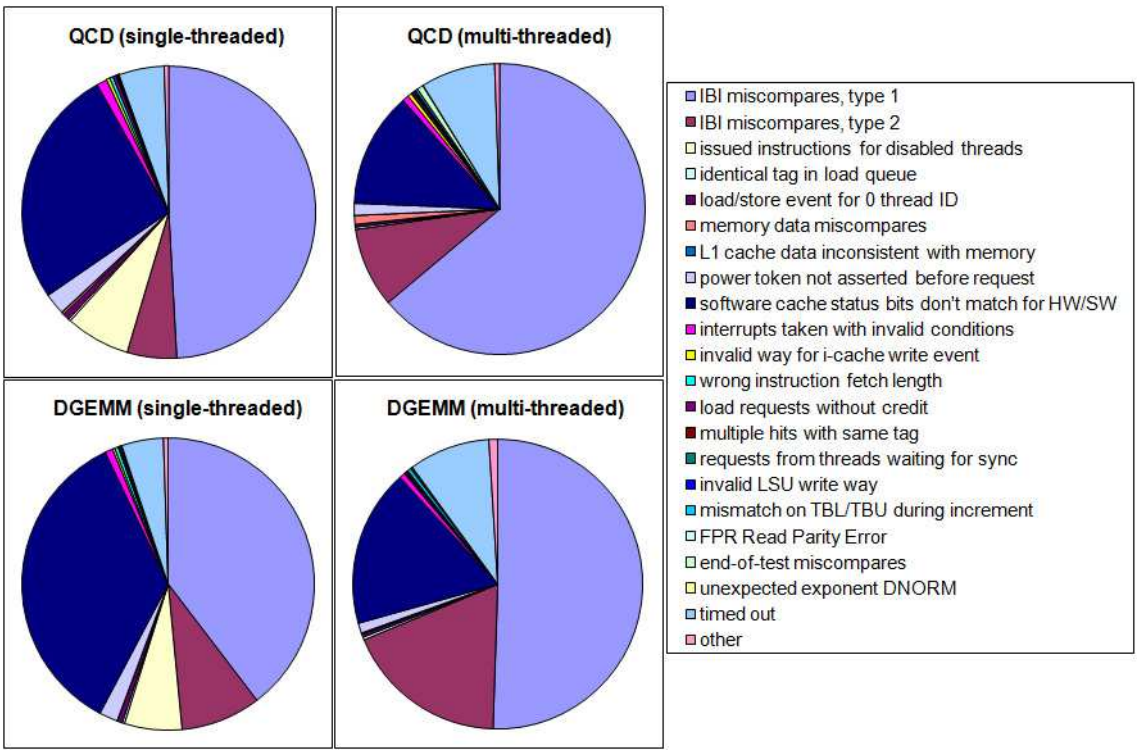

Figure 3.9: Failure Type Distribution

they are one and the same: they are both errors found in architectural state. The figure shows that most faults manifest as IBI miscompares. The data still contain a few SPRs that are not targeted for a hardened design style and that are clocked at the point of injection, contributing somewhat to these state miscompares, but the vast majority of faults here were not injected directly into state. According to these results, most faults end up propagating into state nonetheless.

The second significant failure category involves discrepancies found in the hardware and software caches (shown in dark blue in the figure). This failure type is generically assigned to any cache inconsistency, but the bulk of it is found in valid or lock bits. Interestingly, this error type is far more prominent among single-threaded failures. The single-threaded TSTs are also the only ones issuing instructions for disabled threads, which makes sense; the multi-threaded TSTs utilize all four threads supported by the architecture and will never encounter a disabled one. We can assume that multi-threaded

TSTs also attempt to issue instructions for the wrong thread but cannot detect the problem at issue, since all threads are enabled. Instead, these errors probably manifest as either more state miscompares or hangs, types comprising a greater percentage of total failures in the multi-threaded TSTs.

3.2.3 Failure Rates by Component

Figure 3.10 plots the percentage of runs that fail due to injections in each processor unit, both overall and only among clocked injections. Unlike the charts in previous sections, these contain data weighted by latch width. A_PCQ and BX actually are two of the most vulnerable units, but this is due almost entirely to SPRs, since the fail rates of both units plummet when only active latches are considered. They could both benefit from the more resilient latch design, although whether the installation would be cost effective for such small components is debatable. A_MMQ is the most robust unit, displaying no more than a 3% error rate for any TST. A_XUQ, on the other hand, is a global source of error, with an average fail rate of 10%. IUB and F4B also reveal substantial SERs, although their error rates seem to be much more dependent on the application and number of threads.

Now that we've identified A_XUQ, IUB, and F4B as the most susceptible components, we look more closely at unit internals. Figures 3.11, 3.12, and 3.13 plot the failure rates of each sub-component in those major units, respectively. A_XUQ has a group of unprotected SPRs that fail mostly when not clocked, and several other modules that rarely propagate faults from unclocked injections. Multi-threaded QCD has a more vulnerable load-store unit than any other TST, while multi-threaded DGEMM stresses the fixed-point unit most. XU_CPL is globally the most vulnerable, with LSUCMD close behind. IUB and F4B also display similar results when comparing all injections to only clocked ones, each with the exception of a single component containing unprotected

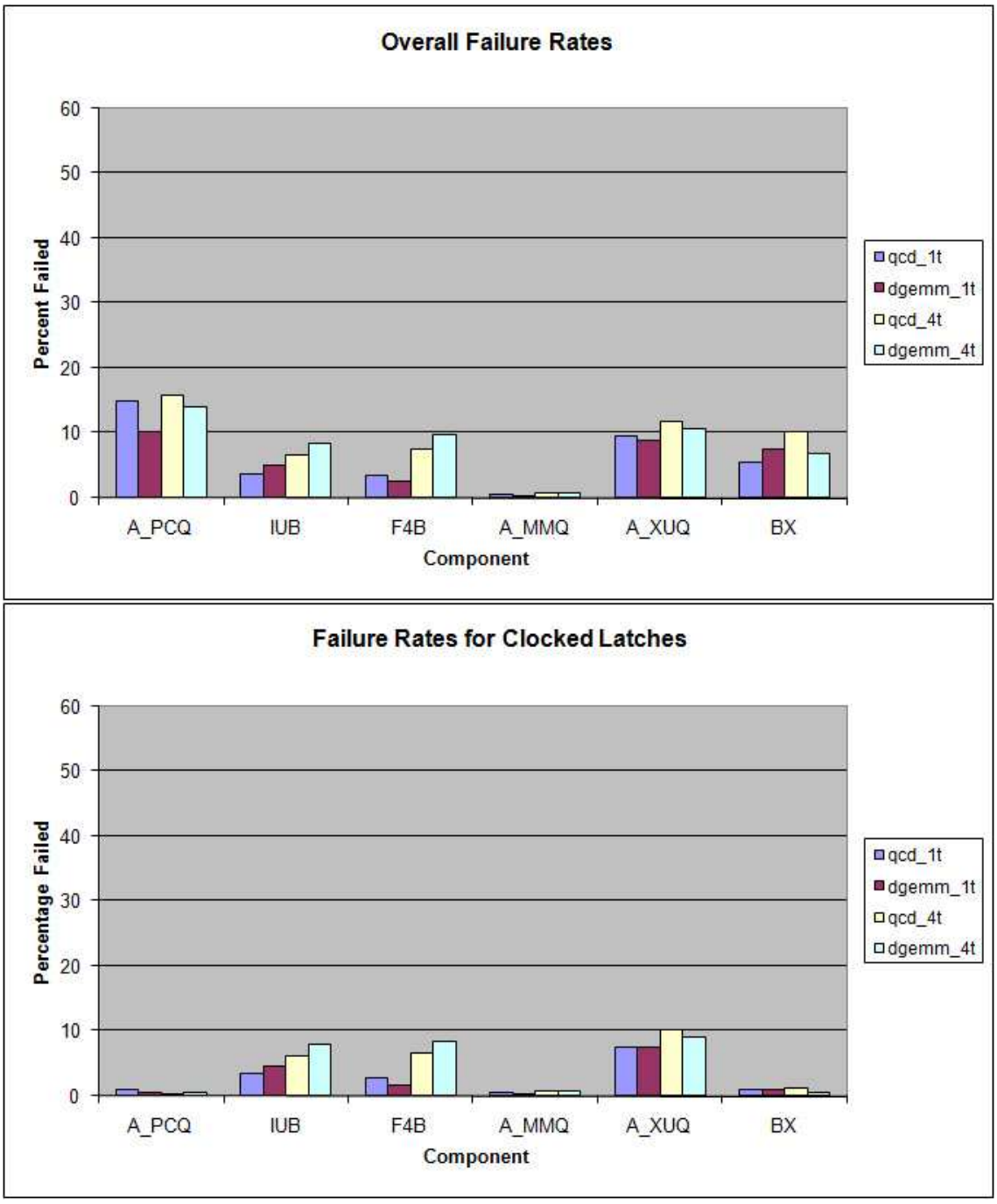

Figure 3.10: Comparison of Unit Failure Rates

SPRs (IUQ_MISC0 for IUB, and DLSP for F4B). In both of these units, multi-threaded DGEMM yields the highest SER vulnerability.

3.2.4 Error Detection Times

Often we wish to know not only where a failure occurs and what error it causes, but how long it takes to discover that error. Cumulative histograms plotting the number of runs that detect errors in each time interval are shown in Figure 3.14. The data for these charts include only those errors detected before program completion, omitting hangs and end-of-test miscompares, and include only clocked injections. The overwhelming majority of errors are detected soon after injection. Eighty percent of failures are discovered within 12 and 25 cycles for single- and multi-threaded DGEMM, and within 49 and 108 cycles for single- and multi-threaded QCD. The data imply that the probability of error discovery at any given point is application dependent. Faults hide in QCD much more readily than they do in DGEMM, and even after 2500 cycles, 5% of QCD errors have yet to surface. Multi-threaded TSTs harbor errors longer than single-threaded ones, due probably in part to the lack of instructions issued for disabled threads.

3.3 Latch Nonuniformity

Thus far we have assumed that all bits within the same latch grouping perform identically, or close to it. There are, however, a few latches in the design that may not conform to this assumption. For instance, floating point data may be subject to rounding, and latch bits carrying the least significant portions of the data will be less vulnerable, since corruptions to those bits will be lost along with the rest of the discarded precision. To test this idea we execute a separate set of simulations, this time injecting only into a single latch suspected of exhibiting nonuniformity. This latch holds the 110-bit result

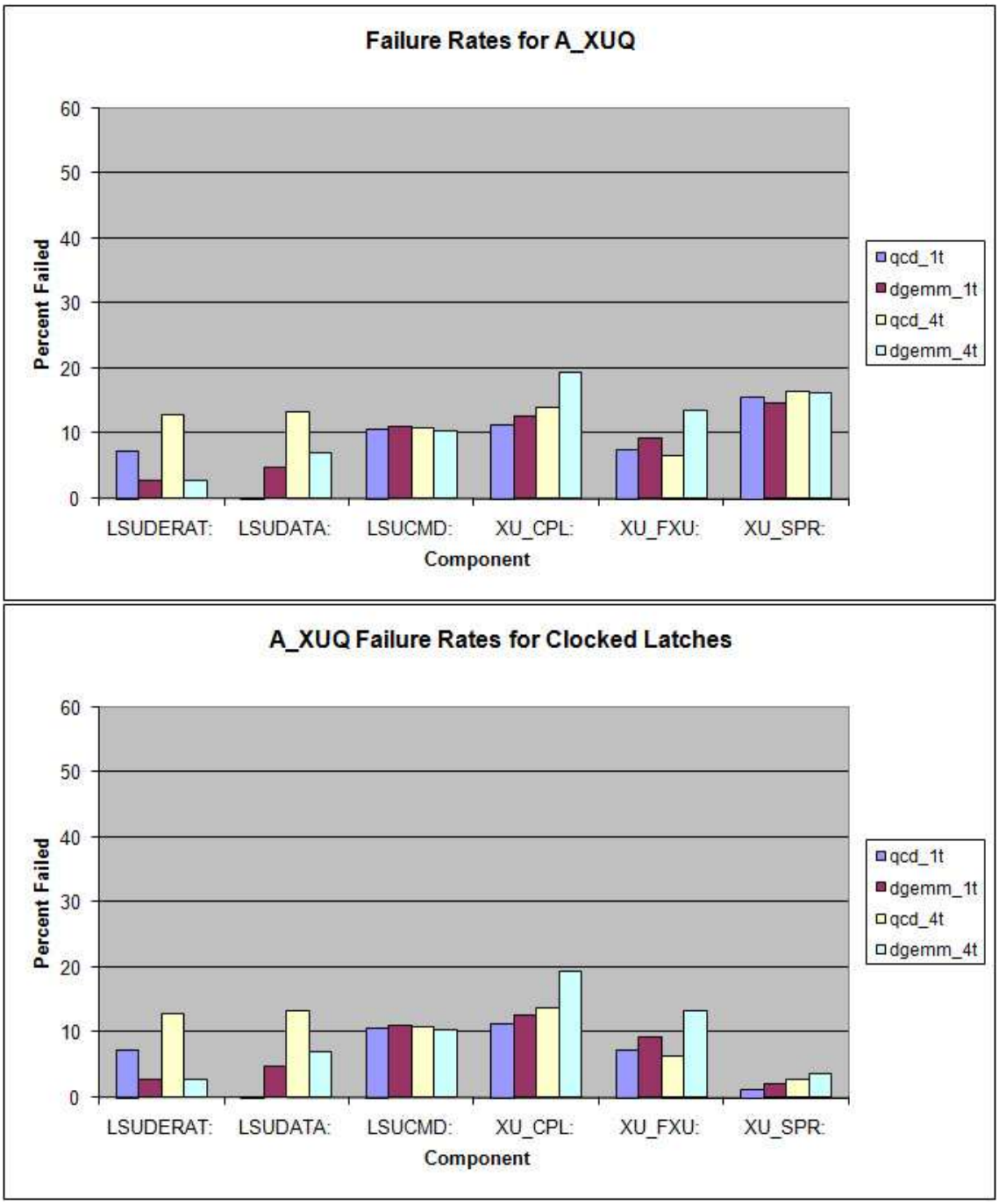

Figure 3.11: Execution Unit Failure Rates

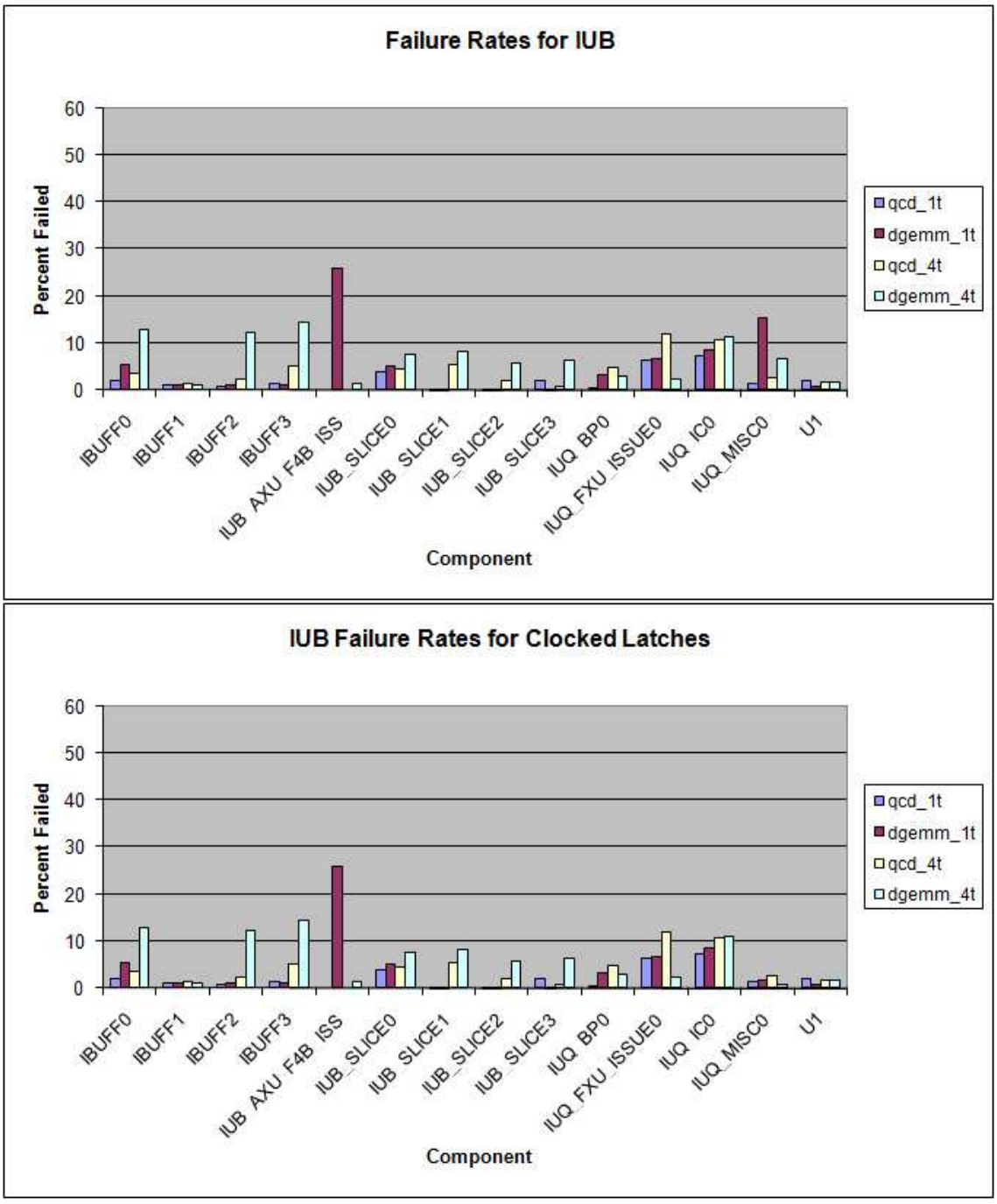

Figure 3.12: Instruction Unit Failure Rates

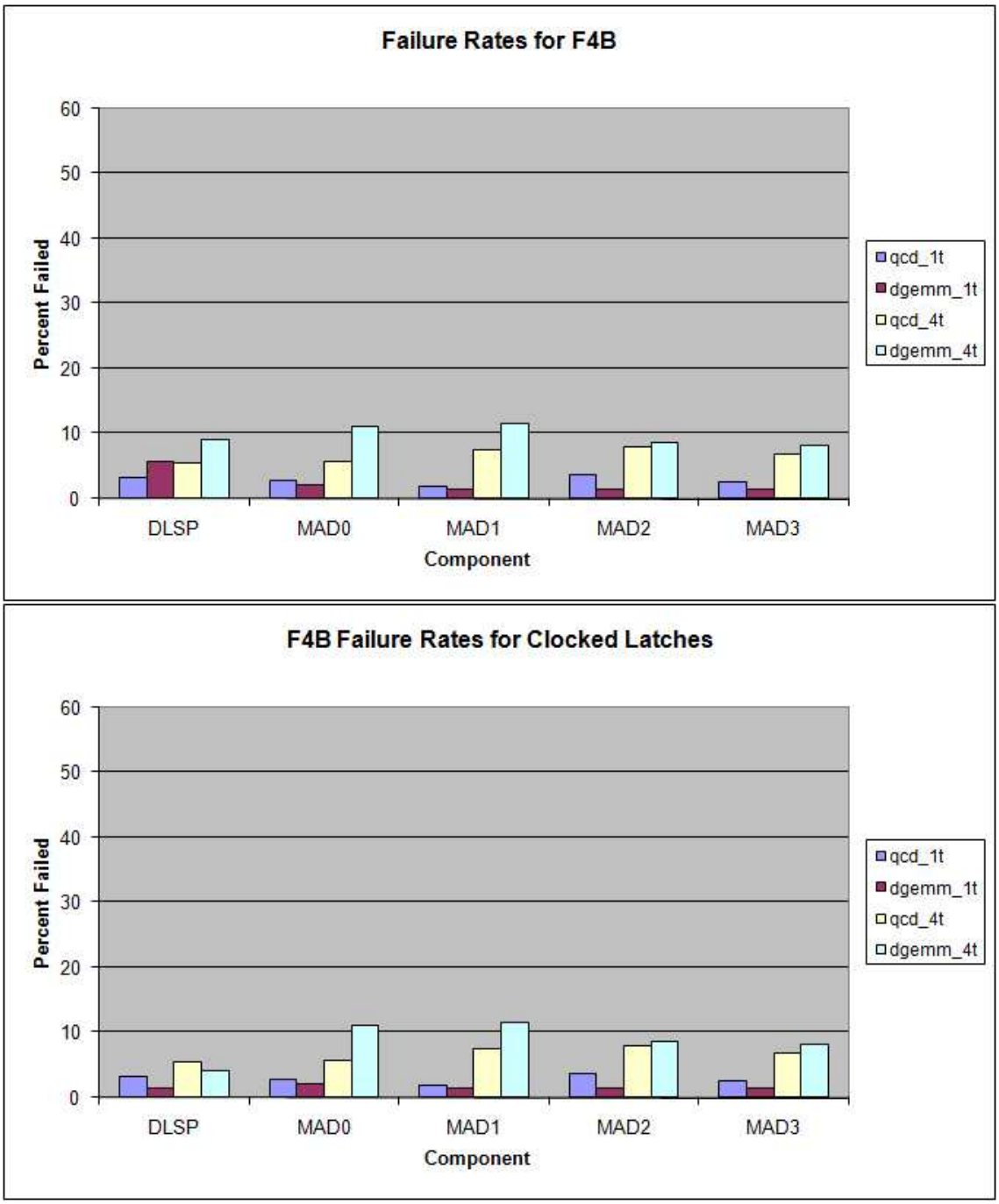

Figure 3.13: Floating Point Unit Failure Rates

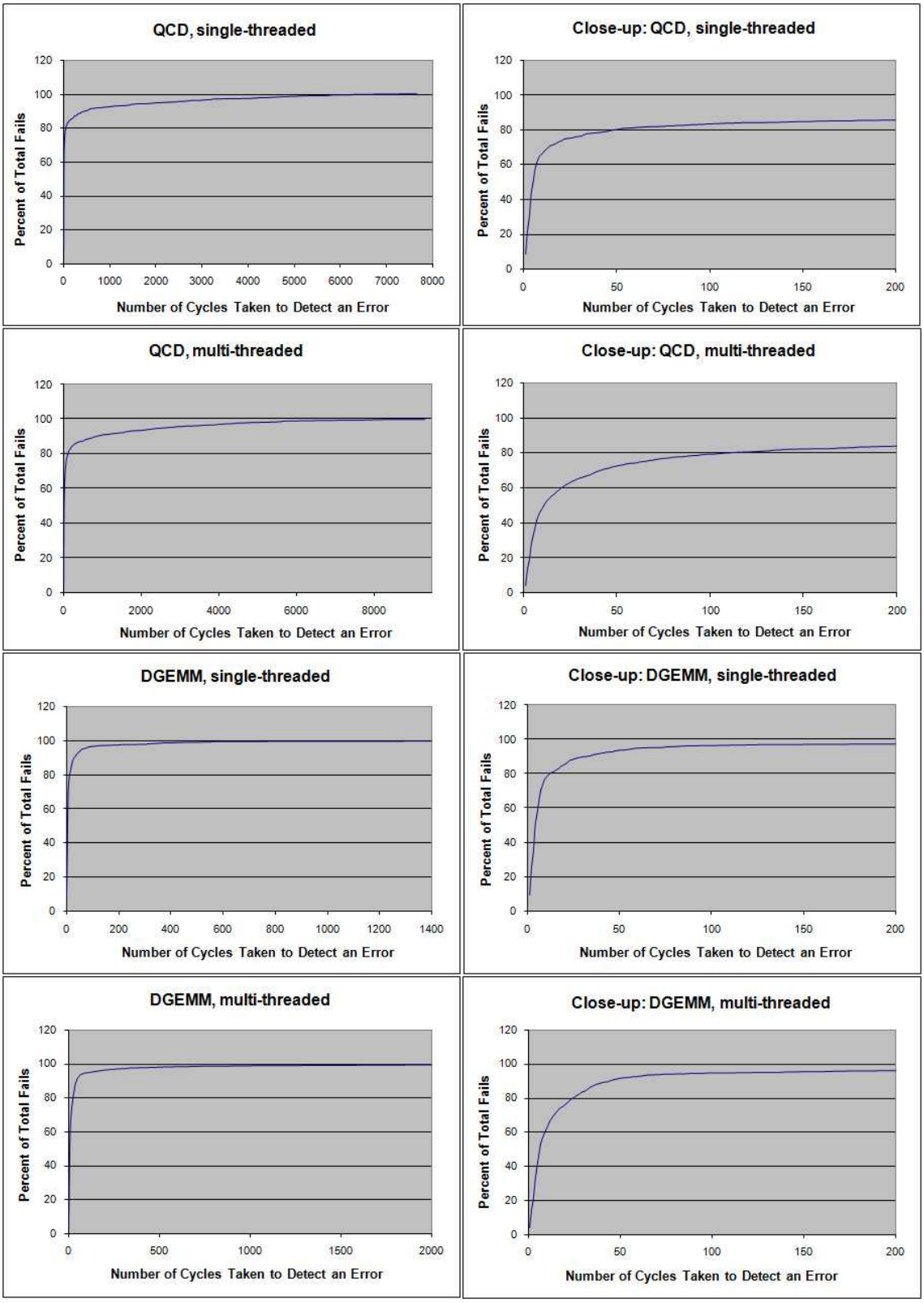

Figure 3.14: Cumulative Histograms: Time from Injection to Detection

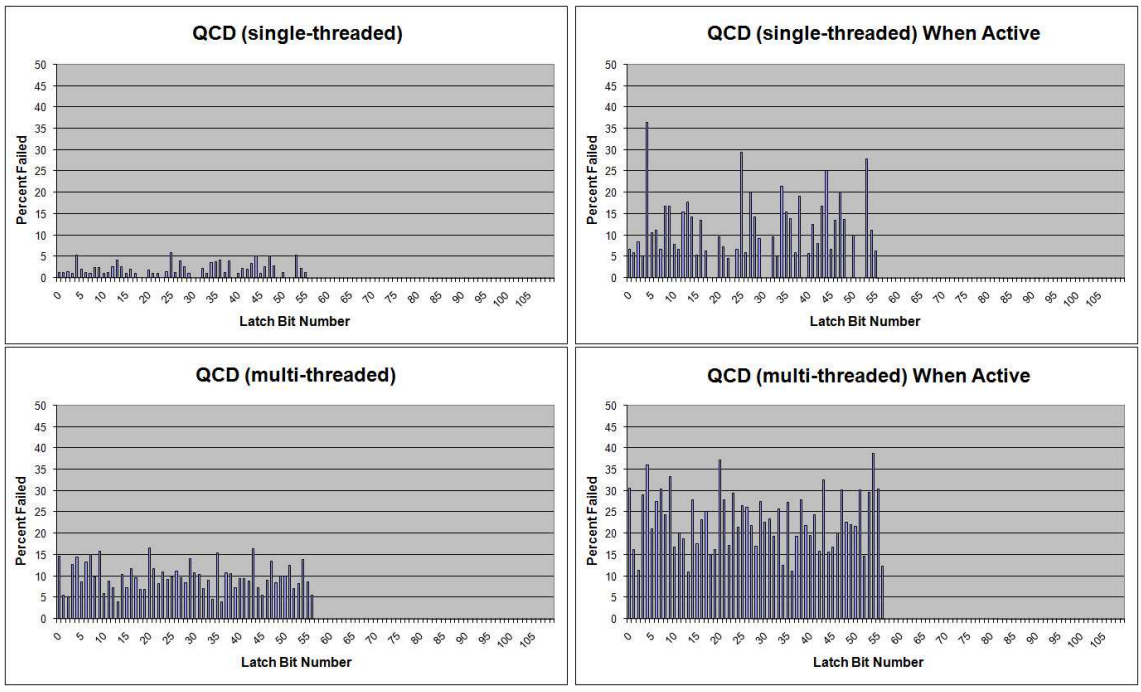

Figure 3.15: Failure Rates Across a Floating Point Latch: F4B.MAD0.FADD.EX4_RES_LO_LAT

of a multiplication operation that is eventually reduced to a 52-bit operand, resulting in a large portion of the latch contributing only to rounding. Approximately 10,000 simulations are run for each of the single- and multi-threaded TSTs.

Figure 3.15 depicts the behavior of a floating point latch heavily used by QCD, both overall and only when clocked (DGEMM does not use this latch often enough to provide data for comparison). Failure rates for each bit in the latch are plotted (zoomed in to 50%) with the most significant bit on the left. The plots show that the SER is unquestionably not uniform across this particular latch type. The most significant half of the latch has, on average, a 1.9% overall / 10.4% clocked error rate for single-threaded QCD and a 9.7% overall / 23.0% clocked error rate for multi-threaded QCD, but the least significant half is completely error-free in all cases. There is a well defined boundary between the two halves of the latch, evidence of where the rounding occurs. This unusual hardware element displays above average susceptibility for half of its data, and based on

our sampling of injections what could be complete immunity to transient faults for the other half. Although latches of this type will not be represented as well by weighting individual injections by latch width, there are only two distinct regions with conflicting behavior, and sufficient sampling of the latch will still capture an accurate approximation of its average SER.

CHAPTER 4

RELATED WORK

We discuss in this section research that others have performed in the areas of fault injection and soft error vulnerability analysis. There have been many studies executed with various implementations and platforms; we mention the most relevant here.

4.1 Fault Injection Studies

Sanda et al. [14] study the soft error resilience of the IBM POWER6 processor using four experiments: proton beam irradiation, fault injection into architected state of the Mambo simulator, statistical fault injection (SFI) into latches of a hardware-emulated RTL model, and neutron beam irradiation. Their goal, like ours, is to determine processor vulnerability, although their target architecture is one designed specifically to be robust against soft errors, while our processor has virtually no existing soft error protection. The SFI experiment, which most closely resembles our own methodology, is described in greater detail by Ramachandran et al. [13]. They obtain greater simulation speed compared to traditional software simulators, but lack the fine-grained error detection capability of RTX.

Wang et al. [18] perform fault injection into latch outputs of an Alpha-based Verilog processor model. They estimate the extent of fault masking and identify vulnerable processor components, and repeat their experiments with some basic protection mechanisms installed. They report numbers similar to ours but use a model developed in an academic environment, representing a processor that does not actually exist, and they also suffer from a lack of detail in their error detection methodology.

In [2], Blome et al. inject faults into a Verilog model of an ARM processor. Their framework supports injection into both sequential and combinational elements, and they

conduct an analysis of error propagation with respect to both types. Instead of identifying error types and error-prone processor locations as we do, they focus more on logical masking rates and general error propagation behavior.

Bronevetsky and de Supinski [3] study the effects of fault injection on iterative linear algebra methods. They employ a high-level technique that injects faults into the data structures of massively parallel applications, and classify errors based on program output. They also propose checkpoint-based fault tolerance mechanisms and perform analyses on the accompanying overheads. They do not utilize a latch accurate simulation infrastructure, and instead focus on much larger scale applications, resulting in work that inherently resides at a higher abstraction level than ours.

Fault injection is sometimes performed at the software level, which can be less expensive but also less precise than hardware fault injection. FERRARI by Kanawati et al. [4] and JIFI by Some et al. [15] are two examples of tools that use software fault injection to research fault tolerance via corruption of the process control structure.

Physical fault injection via particle bombardment or radiation is another alternative methodology. This technique requires the use of expensive resources that few can obtain, and error observation is limited to a very high level. However, of all fault injection techniques it most accurately mimics an authentic soft-error environment and, unlike software and hardware simulation, provides raw failure probability numbers. Lidén et al. [10] perform one such physical fault injection study using heavy-ion circuit radiation to quantify the probability of transient error manifestation in memory elements.

4.2 Processor Vulnerability Estimation

Some research has been dedicated to finding the architectural vulnerability factors (AVF) of various structures in a processor [11] [1] [16]. These studies assess the probability of architectural fault masking through statistical analysis or ACE determination (iden-

tifying bits of state that must be correct for architecturally correct execution). Wang et al. [17] again use fault injection to quantify the conservatism of ACE analysis.

Others have developed their own methodologies for the analytical assessment of SER. Li et al. [9] propose SoftArch, a tool for estimating the mean time to failure (MTTF) of an architecture using probabilistic models for error generation and propagation. Zhang and Shanbhag [19] introduce SERA, a soft error rate analysis approach using conditional probabilities extracted from circuit simulations with applied graph theory. Nguyen and Yagil [12] describe an approach using the timing and logic derating of a circuit to estimate the FIT rate of each processor element.

CHAPTER 5

CONCLUSIONS

The threat of transient fault pollution is ubiquitous and must be addressed by modern processor architects. We have presented a fault injection methodology uniquely capable of capturing the microarchitectural behavior of soft errors at a detailed level, and have demonstrated its results in a new IBM processor core. We utilize a verification tool combined with four TST applications of various complexities to observe the behavior of our host architecture in the presence of individual soft errors at latch outputs. Less than 20% of all faults are found to propagate into user-visible errors for any given TST. When taking into consideration the fact that a few latches have already been designed to be resilient against soft errors and removing these from our dataset, reliability improves by an average of 4%. State miscompares are the most common type of failure, followed by inconsistencies in the hardware and software caches. We determine that the execution unit is the most vulnerable processor component, although the exact source of susceptibility varies with TST. Multi-threaded TSTs in general yield higher failure rates than single-threaded ones in almost every category. The majority of errors are detected shortly after injection, although the percentage of total runs that pass at any given time is application dependent. Finally, we investigate the behavior of a single floating point latch and find that rounding artifacts create fail rate nonuniformity.

To improve reliability for this processor, SPRs and other state latches should be protected, either by changing the physical characteristics of the latches themselves or by employing error detection or correction techniques such as parity or ECC. ECC is a high overhead solution and perhaps not worthwhile for smaller latches, but a larger technology decreases performance. If the decision to improve latch resilience is to be made anywhere, however, it should be made for SPRs, to avoid permanent fault manifestation whenever possible. In addition, clock gating should be universally installed in every unit

so that errors are not unintentionally propagated when the latches are not in use. This has the obvious additional advantage of reducing power consumption. The execution unit should be a primary target, especially because it contains the largest number of latches in the design. The instruction unit and floating point unit have room for improvement as well, and both should be fully investigated for maximum processor robustness.

BIBLIOGRAPHY

[1] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing Architectural Vulnerability Factors for Address-Based Structures. In *Proc. of the International Symposium on Computer Architecture*, Madison, WI, June 2005.

[2] J. Blome, S. Mahlke, D. Bradley, and K. Flautner. A Microarchitectural Analysis of Soft Error Propagation in a Production-Level Embedded Microprocessor. In *Proc. of the First Workshop on Architectural Reliability*, Barcelona, Spain, Nov. 2005.

[3] G. Bronevetsky and B. R. de Supinski. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *Proc. of the International Conference on Supercomputing*, Island of Kos, Greece, June 2008.

[4] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, Feb. 1995.

[5] T. Karnik, P. Hazucha, and J. Patel. Characterization of soft errors caused by single even upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143, Apr. 2004.

[6] T. Karnik, S. Vangal, V. Veeramachaneni, P. Hazucha, V. Erraguntla, and S. Borkar. Selective node engineering for chip-level soft error rate improvement. *Symposium on VLSI Circuits*, pages 204–205, June 2002.

[7] Y. Komatsu, Y. Arima, T. Fujimoto, T. Yamashita, and K. Ishibashi. A Soft-Error Hardened Latch Scheme for SoC in a 90nm Technology and Beyond. In *Proc. of the IEEE Custom Integrated Circuits Conference*, Oct. 2004.

[8] S. Krishnamohan and N. R. Mahapatra. Analysis and Design of Soft-Error Hardened Latches. In *Proc. of the ACM Great Lakes Symposium on VLSI*, Chicago, IL, Apr. 2005.

[9] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors. In *Proc. of the IEEE International Conference on Dependable Systems and Networks*, Yokohama, Japan, June 2005.

[10] P. Lidén, P. Dahlgren, R. Johansson, and J. Karlsson. On Latching Probability of Particle Induced Transients in Combinational Networks. In *Proc. of the International Symposium on Fault-Tolerant Computing*, Austin, TX, June 1994.

[11] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture*, San Diego, CA, Dec. 2003.

[12] H. T. Nguyen and Y. Yagil. A Systematic Approach to SER Estimation and Solutions. In *Proc. of the IEEE International Reliability Physics Symposium*, Dallas, TX, Apr. 2003.

[13] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda. Statistical Fault Injection. In *Proc. of the IEEE International Conference on Dependable Systems and Networks*, Anchorage, AK, June 2008.

[14] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones. Soft-error resilience of the IBM POWER6 processor. *IBM Journal of Research and Development*, 52(3):275–284, May 2008.

[15] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal, and J. J. Beahan. A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance. In *Proc. of the IEEE International Conference on Dependable Systems and Networks*, Goteborg, Sweden, July 2001.

[16] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic Prediction of Architectural Vulnerability from Microarchitectural State. In *Proc. of the International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[17] N. J. Wang, A. Mahesri, and S. J. Patel. Examining ACE Analysis Reliability Estimates Using Fault Injection. In *Proc. of the International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[18] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *Proc. of the IEEE International Conference on Dependable Systems and Networks*, Florence, Italy, July 2004.

[19] M. Zhang and N. R. Shanbhag. A soft error rate analysis (SERA) methodology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2140–2155, Oct. 2006.

[20] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M.

Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40(1):3–18, Jan. 1996.